FORSCHUNGSZENTRUM JÜLICH GmbH
Zentralinstitut für Angewandte Mathematik
D-52425 Jülich, Tel. (02461) 61-6402

Interner Bericht

Parallelizing Iterative Solvers for Sparse Systems of Equations and Eigenproblems on Distributed Memory Machines

Achim Basermann

KFA-ZAM-IB-9411

Mai 1994
(Stand 31.05.94)

Diese Arbeit ist zur Publikation eingereicht.

Parallelizing Iterative Solvers for Sparse Systems of Equations and Eigenproblems on Distributed Memory Machines

A. Basermann *

Central Institute for Applied Mathematics
Research Centre Jülich GmbH, 52425 Jülich, Germany
email: a.basermann@kfa-juelich.de

Abstract

For the analysis and solution of discretized ordinary or partial differential equations it is necessary to solve systems of equations or eigenproblems with coefficient matrices of different sparsity patterns, depending on the discretization method. In many cases, the use of the finite element method (FE) results in largely unstructured systems of equations. The main computational cost in iterative methods for solving these problems consists of matrix-vector products and vector-vector operations; usually the main work in each iteration is the computation of matrix-vector products. When iterative solvers are parallelized on a multiprocessor system with distributed memory, the data distribution and the communication scheme – depending on the data structures used for sparse matrices – are of the greatest importance for an efficient execution. Here, data distribution and communication schemes are presented that are based on the analysis of the column indices of the non-zero matrix elements. Performance tests, using the conjugate gradient method (CG) and the Lanczos algorithm for the symmetric eigenproblem, were carried out on the distributed memory systems INTEL iPSC/860 and PARAGON XP/S 10 of the Research Centre Jülich with sparse matrices from FE models. The parallel variants of the algorithms showed good scaling behavior for matrices with very different sparsity patterns.

Keywords: Sparse matrices; Conjugate gradient method; Lanczos algorithm; Parallelization; Distributed memory computer; Data distribution; Communication scheme.

1 Introduction

For the analysis and solution of discretized ordinary or partial differential equations it is necessary to solve systems of equations or eigenproblems with coefficient matrices of different sparsity patterns, depending on the discretization method. In many cases, the use of the finite element method (FE) results in largely unstructured systems of equations.

Sparse eigenproblems play an important role in the analysis of elastic solids and structures [13] [20] [27]. In the corresponding FE models, the natural frequencies and mode shapes of free vibration are determined, as are buckling loads and modes. Another class of problem is related to stability analysis, e.g. of electrical networks. Moreover, approximations of extreme eigenvalues are useful to solve sets of linear equations, e.g. for the determination of condition numbers of symmetric positive definite matrices or for conjugate gradient methods with polynomial preconditioning [8].

The main computational cost in iterative methods for solving linear systems and eigenproblems consists of matrix-vector products and vector-vector operations; usually the main

*The author was supported by the EEC ESPRIT Basic Research Project 6634 (APPARC) and by the Graduiertenkolleg "Informatik und Technik", RWTH Aachen, Germany.

work in each iteration is the computation of matrix-vector products. The access to the vector is determined by the sparsity pattern and the storage scheme of the matrix.

When iterative solvers are parallelized on a multiprocessor system with distributed memory, the data distribution and the communication scheme – depending on the data structures used for sparse matrices – are of the greatest importance for an efficient execution. In this context, we investigate different reordering strategies of the sparse matrix to reduce waiting times by the overlapped execution of computation and communication. Additionally, the reverse Cuthill-McKee scheme [29] is applied to diminish the bandwidth of the matrix. Depending on the sparsity pattern of the matrix, bandwidth reduction can result in a considerable decrease of communication. The data distribution and the communication scheme are determined before the execution of the solver by preprocessing the symbolic structure of the sparse matrix, and they both are exploited in each iteration. The schemes can be reused as long as the sparsity pattern of the matrix (which is determined by the discretization mesh and the element types) does not change. For example, they can be used in each time step of a time dependent problem or in each iterative step of a nonlinear problem that is solved by linearization. In this report, we present data distribution and communication schemes that are based on the analysis of the column indices of the non-zero matrix elements.

Performance tests, using the conjugate gradient algorithm (CG) with preconditioning [11] [19] [24] to solve systems of equations and the Lanczos method for the symmetric eigenproblem [14] [15] [25] [31] [32], were carried out on the distributed memory systems INTEL iPSC/860 and PARAGON XP/S 10 of the Research Centre Jülich with sparse matrices from two FE models. The first FE model comes from environmental science; it simulates the behavior of pollutants in geological systems [5] [30]. In the second FE model, coming from structural mechanics, stresses in materials induced by thermal expansion are calculated by applying the FE program SMART [6].

The remainder of this report is organized as follows: Section 2 briefly describes two iterative solvers, the CG method and the Lanczos method for the symmetric eigenproblem. In section 3, a conventional storage scheme for large sparse matrices is presented. For the parallelization of iterative methods on a distributed memory system, section 4 gives a detailed description of data distribution and communication schemes based on the storage technique above. In section 5, we discuss numerical and performance results of parallel CG and Lanczos methods on the iPSC/860 and PARAGON XP/S 10, and finally, section 6 is devoted to concluding remarks.

2 Iterative Solvers

In many applications, Krylov subspace methods are applied to solve systems of linear equations or eigenproblems. In the following, we consider some variants of two frequently used algorithms, the method of conjugate gradients and the Lanczos method for the symmetric eigenproblem.

2.1 The Method of Conjugate Gradients

The method of conjugate gradients [19] is an algorithm for solving systems of linear equations $Ax = b$, particularly when A is a sparse coefficient matrix. The method applies to symmetric positive definite matrices $A \in \mathbb{R}^{n \times n}$.

In 1952, Hestenes and Stiefel developed the original CG method that is described in Algorithm 2.1. In each iteration the vectors x_i, g_i, and d_i are computed. x_i approximates the solution vector, g_i is the residual, and d_i determines the direction in which the next approximation of the solution vector is sought. For some sparse matrices, the main work

in each iteration consists solely of the computation of the matrix-vector product Ad_i; for other sparse matrices, this work is comparable with the work involved in the computation of inner products and saxpys. Iteration is continued until the Euclidean norm of the residual is less than or equal to ϵ_r.

Algorithm 2.1. The original CG method

Choose an arbitrary $x_0 \in \mathbb{R}^n$;

$$\begin{aligned} g_0 &= Ax_0 - b \\ d_0 &= -g_0 \end{aligned}$$

$i = 0, 1, \ldots$

$$\begin{aligned} \gamma_i &= \frac{g_i^T g_i}{d_i^T A d_i} \\ x_{i+1} &= x_i + \gamma_i d_i \\ g_{i+1} &= g_i + \gamma_i A d_i \\ \delta_i &= \frac{g_{i+1}^T g_{i+1}}{g_i^T g_i} \\ d_{i+1} &= -g_{i+1} + \delta_i d_i \end{aligned}$$

until $\|g_{i+1}\|_2 \leq \epsilon_r$.

Another stopping criterion that uses the maximum scaled absolute difference of the components of the solution vector's latest two approximations is determined as follows:

$$\max_{j=1,\ldots,n} 2\frac{|x_{i+1}^j - x_i^j|}{|x_{i+1}^j| + |x_i^j|} \leq \epsilon_s . \tag{2.1}$$

In 1990, Aykanat et al. [9] suggested a modified CG algorithm, which has better parallelization properties than the original method.

Algorithm 2.2. The modified CG method

Choose an arbitrary $x_0 \in \mathbb{R}^n$;

$$\begin{aligned} g_0 &= Ax_0 - b \\ d_0 &= -g_0 \end{aligned}$$

$i = 0, 1, \ldots$

$$\begin{aligned} \gamma_i &= \frac{g_i^T g_i}{d_i^T A d_i} \\ \delta_i &= \frac{\gamma_i (Ad_i)^T A d_i}{d_i^T A d_i} - 1 \\ g_{i+1}^T g_{i+1} &= \delta_i g_i^T g_i \\ x_{i+1} &= x_i + \gamma_i d_i \\ g_{i+1} &= g_i + \gamma_i A d_i \\ d_{i+1} &= -g_{i+1} + \delta_i d_i \end{aligned}$$

until $\|g_{i+1}\|_2 \leq \epsilon_r$.

The main difference between the original and the modified algorithm is that in the modified version all dot products are computed without any operations in between. Therefore, if each iteration is performed in parallel on a distributed memory system, the local values of the dot products can be included in one message to determine the global values.

Algorithm 2.2 was shown to be less robust than Algorithm 2.1 for some test matrices. This disadvantage can be avoided by computing the dot product $g_i^T g_i$ in each iteration and not using the value of $\delta_{i-1} g_{i-1}^T g_{i-1}$ from the previous iteration. This results in an additional dot product at the beginning of each iteration, but does not affect the more advantageous parallelization properties of Algorithm 2.2 compared with Algorithm 2.1. With that modification, Algorithm 2.2 shows the same robustness as Algorithm 2.1 for all matrices tested.

In the investigations, Algorithms 2.1 and 2.2 were performed with and without diagonal scaling [24], a simple preconditioner, which hardly contributes to the total execution time but usually accelerates the convergence considerably.

2.2 The Lanczos algorithm

Let A be an $n \times n$ real symmetric matrix with eigensolutions (λ_i, z_i), $i = 1, 2, \ldots, n$, i.e.,

$$Az_i = \lambda_i z_i, \quad i = 1, 2, \ldots, n.$$

For solving real symmetric eigenproblems, Lanczos methods are most commonly used to approximate a small number of extreme eigenvalues and eigenvectors of large sparse matrices [14] [15] [25] [31] [32]. Starting with the original sparse matrix, a sequence of symmetric tridiagonal matrices is generated; the eigenvalues of the tridiagonal matrices approximate the eigenvalues of the original matrix.

2.2.1 Tridiagonalization

The following shows the variant of the Lanczos tridiagonalization suggested in [10].

Algorithm 2.3. The Lanczos tridiagonalization

Choose an arbitrary q_1 *with* $\|q_1\|_2 = 1$ *and set* $q_0 = 0$, $\beta_1 = 0$;

$i = 1, 2, \ldots$

$$\begin{aligned} \alpha_i &= q_i^T A q_i \\ r_i &= A q_i - \beta_i q_{i-1} - \alpha_i q_i \\ \beta_{i+1} &= \|r_i\|_2 \\ q_{i+1} &= \frac{r_i}{\beta_{i+1}} \end{aligned}$$

In this method, a vector sequence $\{q_i\}_{i=1,2,\ldots}$ with $q_i \in \mathbf{R}^n$ and a sequence of $i \times i$ symmetric tridiagonal matrices T_i, $i = 1, 2, \ldots$, are generated by an iterative process starting with an $n \times n$ real symmetric matrix A and an initial vector q_1. The orthonormal vectors q_i, $i = 1, 2, \ldots$, are called the Lanczos vectors and the symmetric tridiagonal matrices T_i the Lanczos matrices. The matrices T_i have the following form with α_m, $m = 1, 2, \ldots, i$,

and β_m, $m = 2, 3, \ldots, i$, as the diagonal and bidiagonal elements, respectively:

$$T_i = \begin{pmatrix} \alpha_1 & \beta_2 & 0 & \cdots & 0 \\ \beta_2 & \alpha_2 & \beta_3 & \cdots & 0 \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ 0 & \cdots & \beta_{i-1} & \alpha_{i-1} & \beta_i \\ 0 & \cdots & 0 & \beta_i & \alpha_i \end{pmatrix}.$$

In Algorithm 2.4, $\|r_i\|_2 = (r_i^T r_i)^{1/2}$ denotes the Euclidean norm of the residual vector $r_i \in \mathbb{R}^n$. The main work in each iteration is the computation of the matrix-vector product Aq_i and, in some situations, also the vector-vector operations.

Because of rounding errors, the Lanczos vectors $q_i \in \mathbb{R}^n$, $i = 1, 2, \ldots$, loose their mutual orthogonality as the number of steps increases. One way to obtain good numerical accuracy is to augment the algorithm with an expensive reorthogonalization step [14] [25]. However, the original Lanczos method, if permitted to continue long enough, is usually able to produce very accurate approximations to all of the distinct eigenvalues of A [10] [14]. That means finite-precision arithmetic delays but it does not prevent the determination of the eigenvalues. Therefore, we do not consider reorthogonalization in this report.

The principle of the modified Lanczos tridiagonalization from [21] is described in the following algorithm. This variant shows more advantageous parallelization properties than Algorithm 2.3.

Algorithm 2.4. The modified Lanczos tridiagonalization

Choose an arbitrary r_0 with $r_0 \neq 0$ and set $q_0 = 0$;

$i = 1, 2, \ldots$

$$\begin{aligned} \beta_i &= \|r_{i-1}\|_2 \\ \alpha_i &= \frac{r_{i-1}^T A r_{i-1}}{r_{i-1}^T r_{i-1}} \\ q_i &= \frac{r_{i-1}}{\beta_i} \\ r_i &= \frac{A r_{i-1}}{\beta_i} - \beta_i q_{i-1} - \alpha_i q_i \end{aligned}$$

The main difference between Algorithm 2.3 and Algorithm 2.4 is again that in the modified version all dot products are computed without any operations in between. For the matrices tested, no difference with respect to robustness was found between the two variants.

2.2.2 Tridiagonal solver

After a certain number of steps of the Lanczos tridiagonalization, the eigenvalues of the current tridiagonal matrix are computed to decide which of those are good approximations of eigenvalues of the original matrix A. This is repeated until a prespecified number of eigenvalues of A are approximated to a given accuracy.

The eigenvalues of the tridiagonal matrices are determined by a bisection method based on the parallel algorithm ALLEV (ALL EigenValues) [12] that uses the Sturm sequence. First, the eigenvalues are isolated in intervals and then extracted to a predefined accuracy by

a superlinearly convergent zero finder, the Pegasus method. ALLEV applies a parallelization strategy over intervals. For use in the Lanczos method, we modified the algorithm to find all eigenvalues in a given interval.

Moreover, we integrated criteria to decide which eigenvalues of the current tridiagonal matrix are good approximations of eigenvalues of A. The following error estimate holds for the eigenvalue μ_j of T_k and the eigenvalue λ_i of A [10] [25]:

$$|\mu_j - \lambda_i| \leq \beta_{k+1}|y_j^k|.$$

y_j^k denotes the kth component of the eigenvector y_j of the eigenpair (μ_j, y_j). The developed parallel Lanczos method LANSP (LANczos SParse) accepts an eigenvalue μ_j of T_k as being sufficiently accurate if

$$\beta_{k+1}|y_j^k| \leq \epsilon_t|\mu_j| \tag{2.2}$$

for a given tolerance ϵ_t. However, the validity of (2.2) only needs to be checked for a certain subset of the eigenvalues of T_k [14]. Therefore, the eigenvalues of T_k are classified; the classes are marked by a corresponding "MP value" as follows [10] [14]:

1. Spurious eigenvalues ($MP = 0$). Let S_{k-1} denote the tridiagonal matrix obtained by deleting the first row and column of T_k. An eigenvalue of T_k is defined to be spurious if it is a numerically simple eigenvalue of T_k and also an eigenvalue of S_{k-1}. A spurious eigenvalue is often a poor approximation to every eigenvalue of A.

2. Multiple eigenvalues ($MP > 1$). Clusters of eigenvalues, i.e., eigenvalues that are extremely close, are assumed a priori to be accurate approximations to eigenvalues of A.

3. Non-isolated simple eigenvalues ($MP = -1$). If any of the remaining eigenvalues is close to a spurious eigenvalue ("close" here is with respect to a larger tolerance than that used before) then that eigenvalue is marked by -1 and assumed to be accurate.

4. Simple eigenvalues ($MP = 1$). The remaining eigenvalues, and only these, are tested for accuracy according to criterion (2.2). For each such eigenvalue, the corresponding eigenvector of T_k is computed using the LAPACK routine DSTEIN [7].

The parallel algorithm LANSP provides the following options:

1. Determining all eigenvalues.

2. Determining all eigenvalues in a given interval.

3. Determining a specific number of eigenvalues.

4. Determining a specific number of eigenvalues in a given interval.

The current version of LANSP does not compute the corresponding eigenvectors. However, this is possible with only slight modifications because the computation of the eigenvectors of the tridiagonal matrices is integrated. For any eigenvector y_j of T_k

$$u_j = \sum_{i=1}^{k} y_j^i q_i$$

is an approximation of an eigenvector of the original matrix A, i.e., the eigenpair (μ_j, u_j) approximates the eigenpair (λ_i, z_i) of A. The corresponding residual vector is

$$\tilde{r}_j = Au_j - \mu_j u_j.$$

For the determination of approximations of the eigenvectors of A, the Lanczos vectors q_i must be either accumulated in secondary storage or recomputed. In the latter case the values of the α_i and β_i, stored as elements of the tridiagonal matrices, can be reused.

3 Storage Schemes

Storage schemes for large sparse matrices depend on the sparsity pattern of the matrix, the considered algorithm, and the architecture of the computer system used. In the literature, many variants of storage schemes can be found [16] [17] [22] [23] [26] [28].

The storage scheme we consider here is often used in FE programs and is suitable for regular as well as for irregular discretization meshes. It can be found in a similar form in [22], for example. The scheme is illustrated in (3.2) for matrix (3.1).

The non-zeros of matrix (3.1) are stored row-wise in three one-dimensional arrays. a^w contains the values of the non-zeros, a^s the corresponding column indices. In a^z, the position of the beginning of each row in a^w and a^s is stored. The subdivisions in a^w and a^s were added to mark the beginning of a new row. The order of the matrix elements per row in a^w and a^s is different from that in matrix (3.1) since this is usually the case in FE programs due to the assembly of the coefficient matrix from single elements.

$$A = \begin{pmatrix} 20 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 30 & 9 & 0 & 0 & 0 & 0 & 0 \\ 0 & 9 & 40 & 10 & 0 & 0 & 0 & 0 \\ 0 & 0 & 10 & 50 & 11 & 14 & 12 & 18 \\ 0 & 0 & 0 & 11 & 60 & 0 & 17 & 0 \\ 0 & 0 & 0 & 14 & 0 & 70 & 15 & 0 \\ 0 & 0 & 0 & 12 & 17 & 15 & 80 & 0 \\ 0 & 0 & 0 & 18 & 0 & 0 & 0 & 90 \end{pmatrix}. \tag{3.1}$$

$$\begin{aligned} a^w &= (20 \mid 9\ 30 \mid 10\ 9\ 40 \mid 10\ 50\ 18\ 14\ 12\ 11 \mid 11\ 60\ 17 \mid 15\ 14\ 70 \mid 12\ 17\ 80\ 15 \mid 18\ 90), \\ a^s &= (1 \mid 3\ 2 \mid 4\ 2\ 3 \mid 3\ 4\ 8\ 6\ 7\ 5 \mid 4\ 5\ 7 \mid 7\ 4\ 6 \mid 4\ 5\ 7\ 6 \mid 4\ 8), \\ a^z &= (1\ 2\ 4\ 7\ 13\ 16\ 19\ 23\ 25). \end{aligned} \tag{3.2}$$

4 Parallelization

For the parallelization of iterative solvers on a distributed memory system, it is essential to distribute the matrix and vector arrays suitably to each processor and to determine an efficient communication scheme for matrix-vector products depending on the sparsity pattern of the matrix.

4.1 Data Distribution

In the data distribution schemes we consider below, the matrix arrays a^w and a^s are distributed row-wise; the rows of each processor succeed one another. The distribution of the vector arrays corresponds component-wise to the row distribution of the matrix arrays.

Criteria for the data distribution can be: Each processor gets the same number of rows or enough rows to ensure that each processor has nearly the same number of non-zeros. The number of operations for the computation of the matrix-vector product is proportional to the number of non-zeros; the remaining vector operations of one iteration are proportional to the number of rows. The criterion we consider here is that each processor has to compute nearly the same number of operations. If the discretization mesh is regular, i.e., the sparsity pattern of the coefficient matrix is regular, then all three criteria result in nearly the same data distribution. If the mesh is very irregular, the three distributions differ considerably.

Our algorithm for distributing the rows onto the processors can be described as follows. The row distribution is determined by analyzing the array a^z. Let n_k denote the number

of rows assigned to processor k, and let p denote the number of processors. Also, let e_k denote the number of non-zeros assigned to processor k, and let e denote the total number of non-zeros in the matrix. Then processor k is assigned rows until the following requirement is satisfied for the first time:

$$\frac{e_k + \xi n_k}{e + \xi n} \geq \frac{1}{p}, \quad \text{for } e_k, n_k \gg 10. \tag{4.1}$$

First, the parameter ξ depends on the number of vector operations that are additional to the operations of the matrix-vector product in each iteration. Secondly, it considers the execution times of arithmetical, logical, and memory operations on the processor used; it is therefore dependent on the processor architecture. The numerator in (4.1) is proportional to the number of operations of one partial iteration on processor k, the denominator is proportional to the total number of operations of one iteration. It should be noted that for $\xi \to 0$ each processor gets nearly the same number of non-zeros and for $\xi \to \infty$ nearly the same number of rows. The first case means that the execution time of all vector-vector operations is negligible compared with the execution time of the matrix-vector product. In the second case, the execution time of the matrix-vector product hardly contributes to the total execution time.

With these considerations, the contribution of the matrix-vector product to one iteration can be approximated by

$$a_{\text{MVP}} \approx \frac{e}{e + \xi n} = \frac{1}{1 + \xi / m_z}, \quad \text{for } e, n \gg 10. \tag{4.2}$$

Here, $m_z = e/n$ is the mean number of non-zeros per row. Additionally, (4.2) provides a means for measuring ξ. If a_{MVP} is determined by timings, then an approximation of ξ can be computed by

$$\xi \approx m_z \left(\frac{1}{a_{\text{MVP}}} - 1 \right).$$

On the INTEL i860 XR or i860 XP, the timings result in an approximative value ξ of about 8.3 for both variants of the CG method and of about 2 for both variants of the Lanczos tridiagonalization.

The data distribution according to criterion (4.1) is shown in (4.3) by distributing matrix (3.1) to four processors. The other arrays are distributed analogously. In this small example, the data distribution is the same for both the CG method and the Lanczos tridiagonalization. For large sparse matrices from FE applications, the data distribution usually varies for these algorithms due to the different values of ξ.

$$\begin{aligned}
&\text{Processor 0: } a_0^w &&= (20 \mid 9\ 30 \mid 10\ 9\ 40),\\
&\text{Processor 1: } a_1^w &&= (10\ 50\ 18\ 14\ 12\ 11 \mid 11\ 60\ 17),\\
&\text{Processor 2: } a_2^w &&= (15\ 14\ 70 \mid 12\ 17\ 80\ 15),\\
&\text{Processor 3: } a_3^w &&= (18\ 90).
\end{aligned} \tag{4.3}$$

4.2 Communication Schemes

On a distributed memory system, the computation of the matrix-vector product requires communication because each processor owns only a partial vector. For the efficient computation of the matrix-vector product, it is necessary to develop a suitable communication scheme by preprocessing the distributed column index arrays. Here, we describe different schemes based on different reorderings of the matrix.

First, the arrays a_k^s are analyzed on each processor k to determine which data results in access to components of the vector of other processors. Then, a_k^s and a_k^w are reordered

in such a way that the data that results in access to processor h are collected in block h. The data of block h succeeds one another row-wise with increasing column index per row. Block k is the first block in a_k^s and a_k^w and contains the data that results in local access. The goal of this reordering is to perform computation and communication overlapped.

The first reordering scheme is shown in (4.4) for the data distribution from (4.3) and the matrix-vector product Ad_i from Algorithms 2.1 or 2.2. Here, only array a_1^s is analyzed and reordered.

$$\begin{array}{lrcl}
\text{Processor 0:} & a_0^s & = & (1 \mid 3\ 2 \mid 4\ 2\ 3), \quad d_{i,0} = (d_i^1\ d_i^2\ d_i^3) \\
\underline{\text{Processor 1:}} & a_1^s & = & (\boxed{3}\ 4\ \boxed{8}\ \boxed{6}\ \boxed{7}\ 5 \mid 4\ 5\ \boxed{7}), \quad d_{i,1} = (d_i^4\ d_i^5) \\
\text{Processor 2:} & a_2^s & = & (7\ 4\ 6 \mid 4\ 5\ 7\ 6), \quad d_{i,2} = (d_i^6\ d_i^7) \\
\text{Processor 3:} & a_3^s & = & (4\ 8), \quad d_{i,3} = (d_i^8)
\end{array} \tag{4.4}$$

$$\underline{\text{Reordering:}}\ a_1^s = (\underbrace{4\ 5 \mid 4\ 5}_{1} \parallel \underbrace{\boxed{3}}_{0} \parallel \underbrace{\boxed{6}\ \boxed{7} \mid \boxed{7}}_{2} \parallel \underbrace{\boxed{8}}_{3})$$

Computing the operation row-times-vector of the matrix-vector product of processor 1, the index 3 results in an access to component d_i^3 of processor 0, the index 8 to d_i^8 of processor 3, and the indices 6 and 7 in access to d_i^6 and d_i^7 of processor 2. The data blocks in (4.4) are separated by double dashes for elucidation; the blocks were numbered below the brackets. After reordering, the data of block 1 results in local access, the data of block 0 in access to processor 0, the data of block 2 in access to processor 2, and the data of block 3 in access to processor 3.

After having analyzed the column index array a_k^s, each processor k knows which components of d_i are required by which processors. This information is broadcasted to all processors. Then, each processor can distinguish which data must be sent to which processors. This communication scheme is determined once before the start of the parallel CG algorithm or Lanczos tridiagonalization and applies then unchanged to each iteration.

Fig. 4.1 displays the communication scheme for the example above. Processor 1 receives the third component of d_i from processor 0, the sixth and seventh component from processor 2, and the eighth component from processor 3. On the other hand, the fourth component of processor 1 is sent to processor 0, the fourth and fifth to processor 2, and the fourth to processor 3.

In Fig. 4.2, the parallel computation of the matrix-vector product is described for the considered algorithms. First on each processor, the data that is necessary for other processors is sent asynchronously. After having executed asynchronous receive-routines for receiving non-local data, each processor performs all local computations, in particular the local part of the computation of the matrix-vector product. Then each processor waits until the data from an arbitrary processor arrives and continues the computation of the matrix-vector product. Thereafter, each processor awaits the data of other processors until the computation of the matrix-vector product is complete. Computation and communication are performed overlapped. While the required data is on the network, operations with local or with data that has already arrived from other processors are executed.

In the second reordering scheme, the data blocks, built as described above, are sent to the processors that own the corresponding components of the vector of the matrix-vector product. The goal is to increase the number of local computations while the required data is on the network. In this case, the processors compute partial results of the result vector of the matrix-vector product. Then, $y_{k,l}$ denotes the partial result of $y_k = A_k d_i$ of processor k that is computed on processor l. After the computation, the partial results except the local

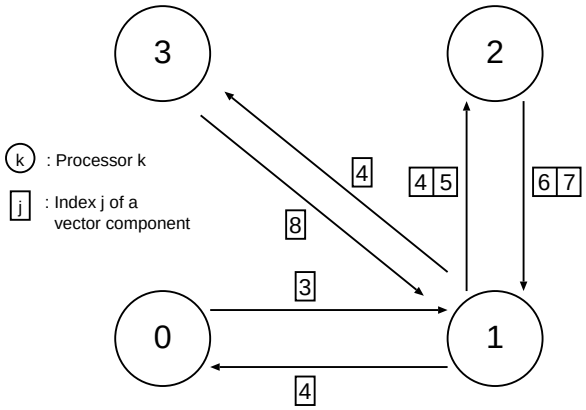

Figure 4.1: Communication scheme, reordering 1

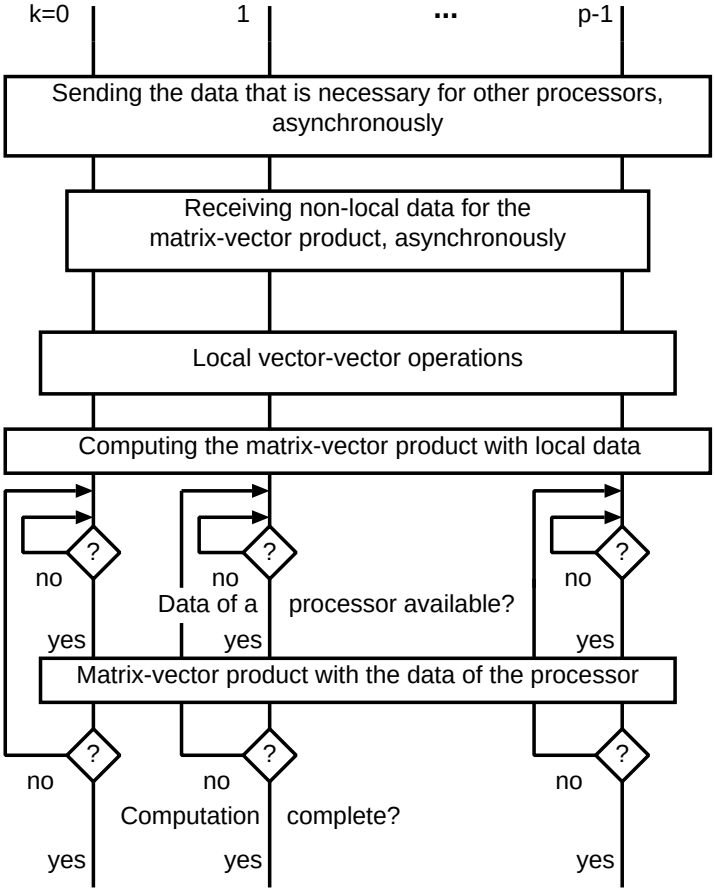

Figure 4.2: The parallel matrix-vector product, reordering 1

one are sent to the corresponding processors and are then added to the local result of the receiving processor. The new distribution of the matrix data is presented in (4.5).

$$
\begin{aligned}
\text{Processor 0: } a_0^s &= (\underbrace{1 \mid 2\ 3 \mid 2\ 3}_{0,0} \parallel \underbrace{\boxed{3}}_{1,0}), \quad d_{i,0} = (d_i^1\ d_i^2\ d_i^3) \\
\text{Processor 1: } a_1^s &= (\underbrace{4\ 5 \mid 4\ 5}_{1,1} \parallel \underbrace{\boxed{4}}_{0,1} \parallel \underbrace{\boxed{4} \mid \boxed{4}\ \boxed{5}}_{2,1} \parallel \underbrace{\boxed{4}}_{3,1}), \quad d_{i,1} = (d_i^4\ d_i^5) \qquad (4.5) \\
\text{Processor 2: } a_2^s &= (\underbrace{6\ 7 \mid 6\ 7}_{2,2} \parallel \underbrace{\boxed{6}\ \boxed{7} \mid \boxed{7}}_{1,2}), \quad d_{i,2} = (d_i^6\ d_i^7) \\
\text{Processor 3: } a_3^s &= (\underbrace{8}_{3,3} \parallel \underbrace{\boxed{8}}_{1,3}), \quad d_{i,3} = (d_i^8)
\end{aligned}
$$

The first number of the blocks in (4.5) denotes the processor to which the partial result is sent; the second number indicates the processor on which the computation is performed. Processor 1, for example, computes the local result $y_{1,1}$ with the first block, the partial result $y_{0,1}$ of processor 0 with the second block, the partial result $y_{2,1}$ of processor 2 with the third block, and the partial result $y_{3,1}$ of processor 3 with the fourth block.

Fig. 4.3 shows the communication scheme for the block distribution from (4.5). Processor 1 sends a value to processor 0, and this value is added to the third component of y_0. Simultaneously, processor 1 receives a value from processor 0 that must be added to the first component of y_1.

In Fig. 4.4, the parallel computation of the matrix-vector product is presented for the second reordering scheme. First, asynchronous receive-routines for receiving all necessary partial results of other processors are executed on each processor. After that, each processor computes the partial results that are sent to other processors. The computation is performed per data block; the results are asynchronously sent to the corresponding processors after each computation. Then, all local computations are performed, in particular the local part of the computation of the matrix-vector product. Thereafter, each processor waits until the data of an arbitrary processor arrives and then adds the values to the corresponding components of the local result. This is repeated until the computation of the matrix-vector product is complete. Computation and communication are performed overlapped.

Since partial results of the matrix-vector product are exchanged most computations are local. After having received non-local data, each processor merely performs a summation of vector components. However, the partial results must be computed first, then they can be sent. That means that the overlap of computation and communication is less than for the first scheme. Another disadvantage of this method is that load balancing is not guaranteed any more after the new distribution of the blocks; some processors can own more or larger data blocks than other ones. However, this scheme allows arbitrary data distribution; each processor can get arbitrary parts of arbitrary rows, which need not succeed one another. For a specific FE application, a suitable data distribution for this scheme can be found when the data distribution considers the discretization mesh.

In the third communication scheme the matrix-vector product is performed column-wise. In this case, the data distribution and the reordering scheme is the same as shown in (4.4), but partial results of the matrix-vector product are sent and received. Compared with the first communication scheme, the data transfer scheme is vice versa: The indices of the components that are sent in the first scheme are received in the third scheme, and the indices of the components that are received in the first scheme are sent. Fig. 4.5 shows the communication scheme for the example above.

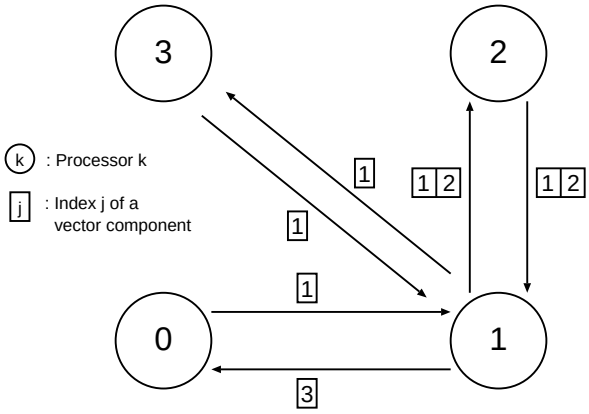

Figure 4.3: Communication scheme, reordering 2

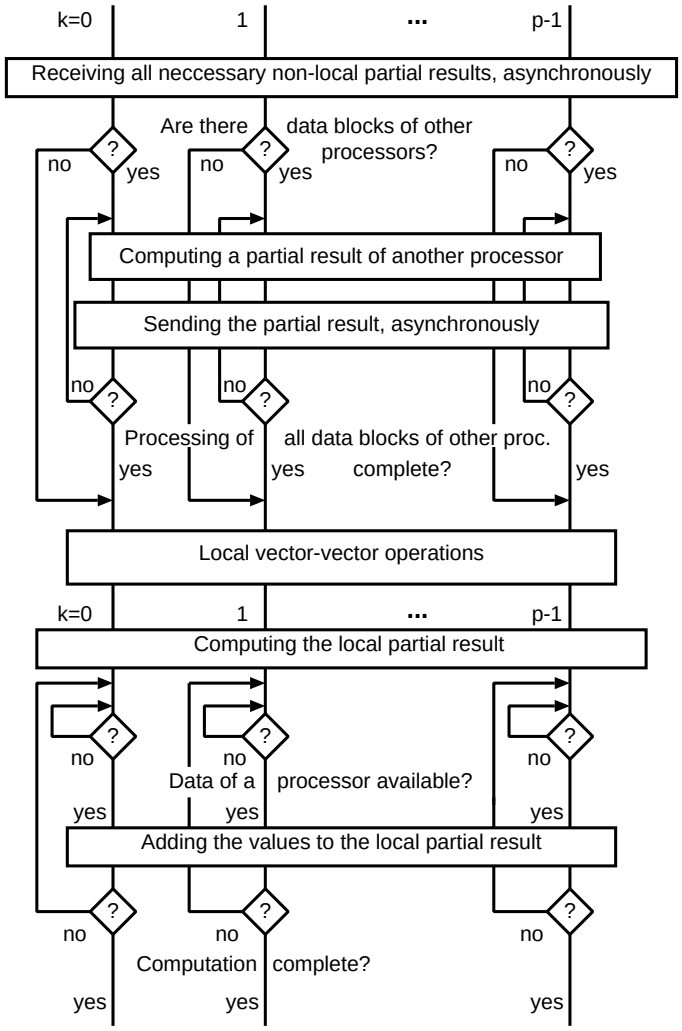

Figure 4.4: The parallel matrix-vector product, reordering 2

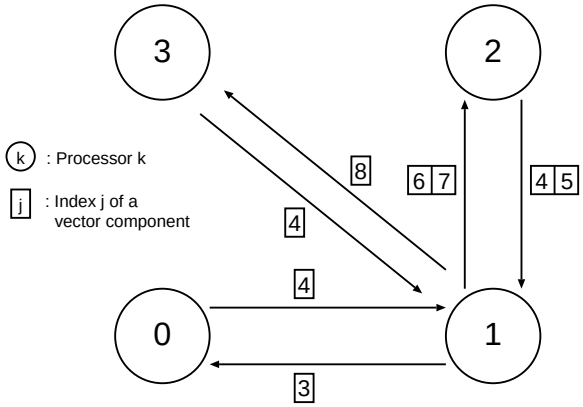

Figure 4.5: Communication scheme 3

Processor 1 sends a partial result of the third component of the global result vector of the matrix-vector product to processor 0, and this value is added to the third component of the local result vector of processor 0. Simultaneously, processor 1 receives a partial result of the fourth component of the global result vector of the matrix-vector product from processor 0 that must be added to the first component of the local result vector of processor 1.

The strategy for the overlapped execution of computation and communication is the same as shown in Fig. 4.4: Partial results must be computed first, then they can be sent. With respect to the overlapped execution, that means a disadvantage compared with the first scheme. However, the number of local computations are increased since, after having received non-local data, each processor merely performs a summation of vector components. Compared with the second scheme, it is advantageous that load balancing is not affected because the data blocks are not redistributed.

The data distribution and the communication schemes we present here do not require any knowledge about a specific discretization mesh; the schemes are determined automatically by the analysis of the column indices of the non-zero matrix elements.

Moreover, these schemes can be applied to other algorithms, such as CG methods with polynomial preconditioning [8] and the QMR algorithm for solving non-Hermitian systems of linear equations [18]. From the operational point of view, polynomial preconditioning results essentially in additional matrix-vector products per CG iteration; the number of the matrix-vector products depends on the degree of the polynomial. In each QMR iteration, two matrix-vector products are performed. With respect to the data distribution, the parameter ξ must be adapted to these algorithms. For CG methods with polynomial preconditioning, the value of the pure CG method divided by the number of matrix-vector products per iteration is usually a sufficient approximation of ξ. With respect to the parallel matrix-vector products, all three communication schemes can be applied. In the case of CG methods with polynomial preconditioning, all matrix-vector products per iteration are coupled, whereas in the QMR iteration, the two matrix-vector products of the form As_i and $A^T t_i$ are decoupled, i.e., $A^T t_i$ is independent of the result of As_i and vice versa. For the matrix-vector product $A^T t_i$, the third communication scheme is most advantageous since if A is stored row-wise then A^T is given column-wise. The same data structure can be accessed for both As_i and $A^T t_i$ if the first scheme is used for As_i. In addition, both schemes can be coupled, i.e., non-local data for As_i and $A^T t_i$ can be included in the same message. The advantage to be achieved is a decreased number of messages and an increased overlap of computation and communication, which makes QMR even more attractive for parallel computing.

5 Results

The numerical and performance tests of the developed parallel CG algorithms and the parallel Lanczos methods were performed on the distributed memory system iPSC/860 and PARAGON XP/S 10 of the Research Centre Jülich. The iPSC/860 has 32 processors, each with a 16 Megabyte private memory, interconnected by a hypercube-network, whereas the PARAGON XP/S 10 has 140 processors, each with a 32 Megabyte private memory, interconnected by a two-dimensional mesh. The maximum transfer rates are 2.8 and 200 Megabyte/second per channel in both directions, respectively.

5.1 Numerical Results

The tests we present here were carried out with one matrix from each of the FE models from environmental science and structural mechanics and in addition with a sample matrix from [14].

The start vector x_0 for Algorithms 2.1 and 2.2 is either given by the FE models or computed by

$$x_0^j = \frac{b_j}{a_{jj}}, \quad j = 1, \ldots, n$$

where $a_{jj} \neq 0$, $j = 1, \ldots, n$, denote the diagonal elements of A. For Algorithm 2.4, the components r_0^j of the start vector r_0 are chosen as

$$r_0^j = \frac{j}{n}, \quad j = 1, \ldots, n.$$

The corresponding start vector q_1 for Algorithm 2.3 is computed by $q_1 = r_0/\sqrt{r_0^T r_0}$, i.e.,

$$q_1^j = \frac{\sqrt{6}j}{\sqrt{n(n+1)(2n+1)}}, \quad j = 1, \ldots, n.$$

These start vectors are advantageous for the parallel execution of the algorithms since they can be generated fully in parallel and are always the same independent of the number of processors used; they were applied in all tests.

As a sample matrix of order $n = 1000000$, we choose matrix (5.1) from [14]. The eigenvalues of the matrix are given by the formula

$$\lambda(i,j) = 1 - \frac{1}{2}\cos\frac{\pi i}{3} - \frac{1}{2}\cos\frac{\pi j}{500001}$$

with $1 \leq i \leq 2$ and $1 \leq j \leq 500000$. For the tests with Algorithms 2.1 and 2.2 using matrix (5.1), the right hand side is chosen as

$$b^j = \sum_{i=1}^{n} s_{ji}, \quad j = 1, \ldots, n$$

so that the exact solution vector is $x = (1, \ldots, 1)^T$.

Table 5.1 shows numerical data of the coefficient matrices and for the convergence of the CG method.

$$S = \begin{pmatrix} 1 & -\frac{1}{4} & -\frac{1}{4} & 0 & 0 & 0 & 0 & \cdots & 0 \\ -\frac{1}{4} & 1 & 0 & -\frac{1}{4} & 0 & 0 & 0 & \cdots & 0 \\ -\frac{1}{4} & 0 & 1 & -\frac{1}{4} & -\frac{1}{4} & 0 & 0 & \cdots & 0 \\ 0 & -\frac{1}{4} & -\frac{1}{4} & 1 & 0 & -\frac{1}{4} & 0 & \cdots & 0 \\ \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \vdots \\ 0 & \cdots & 0 & -\frac{1}{4} & 0 & 1 & -\frac{1}{4} & -\frac{1}{4} & 0 \\ 0 & \cdots & 0 & 0 & -\frac{1}{4} & -\frac{1}{4} & 1 & 0 & -\frac{1}{4} \\ 0 & \cdots & 0 & 0 & 0 & -\frac{1}{4} & 0 & 1 & -\frac{1}{4} \\ 0 & \cdots & 0 & 0 & 0 & 0 & -\frac{1}{4} & -\frac{1}{4} & 1 \end{pmatrix} \tag{5.1}$$

Environmental science		Structural mechanics	Sample matrix
Rows	49392	25222	1000000
Non-zeros	1242814	3856386	3999996
Density	0.05%	0.6%	0.0004%
Non-zeros per row, max.	27	485	4
m_z	25.2	152.9	4.0
a_{MVP}, CG method	75%	95%	33%
a_{MVP}, Lanczos tridiagonalization	93%	99%	67%
Condition number	$\approx 10^3$	$\approx 10^5$	≈ 7
CG method: max. scal. abs. diff. $\leq 10^{-5}$			
Iterations without scaling	390	1444	14
Iterations with scaling	84	658	14
$\|g_{i+1}\|_2$	4.5×10^{-4}	1.5×10^{-5}	1.4×10^{-12}

Table 5.1: Numerical data of the considered large sparse matrices

The matrix from environmental science has 49392 rows, that from structural mechanics 25222. In the first case, the mean number of non-zeros per row is near the maximum number. This is caused by a regular discretization mesh. In the second case, the mean and the maximum number are markedly different; the discretization mesh is much more irregular. The operational contribution of the matrix-vector product to one iteration is 75% for the matrix from environmental science, 95% for the matrix from structural mechanics, and 33% for the sample matrix in the case of the CG method; in the case of the Lanczos tridiagonalization, the values are 93%, 99%, and 67%, respectively. For the first two matrices, the condition number with respect to the spectral norm that is the ratio of the largest and the smallest eigenvalue in the case of symmetric positive definite matrices was approximated by the developed parallel Lanczos method applying Algorithm 2.4.

In Table 5.1, the number of CG iterations with and without diagonal scaling is given. The iteration is stopped when the maximum scaled absolute difference from (2.1) is less than 10^{-5}; this corresponds to a precision of the solution vector of about five decimals. With diagonal scaling, the number of iterations is considerably smaller in the first two cases. In the third case, diagonal scaling is not effective since the diagonal elements are equal to one. The contribution of this preconditioner to the total execution time is in all cases below 1%. For the preconditioned method, the Euclidean norm of the residual after 84, 658, and 14 iterations, respectively, is given in addition.

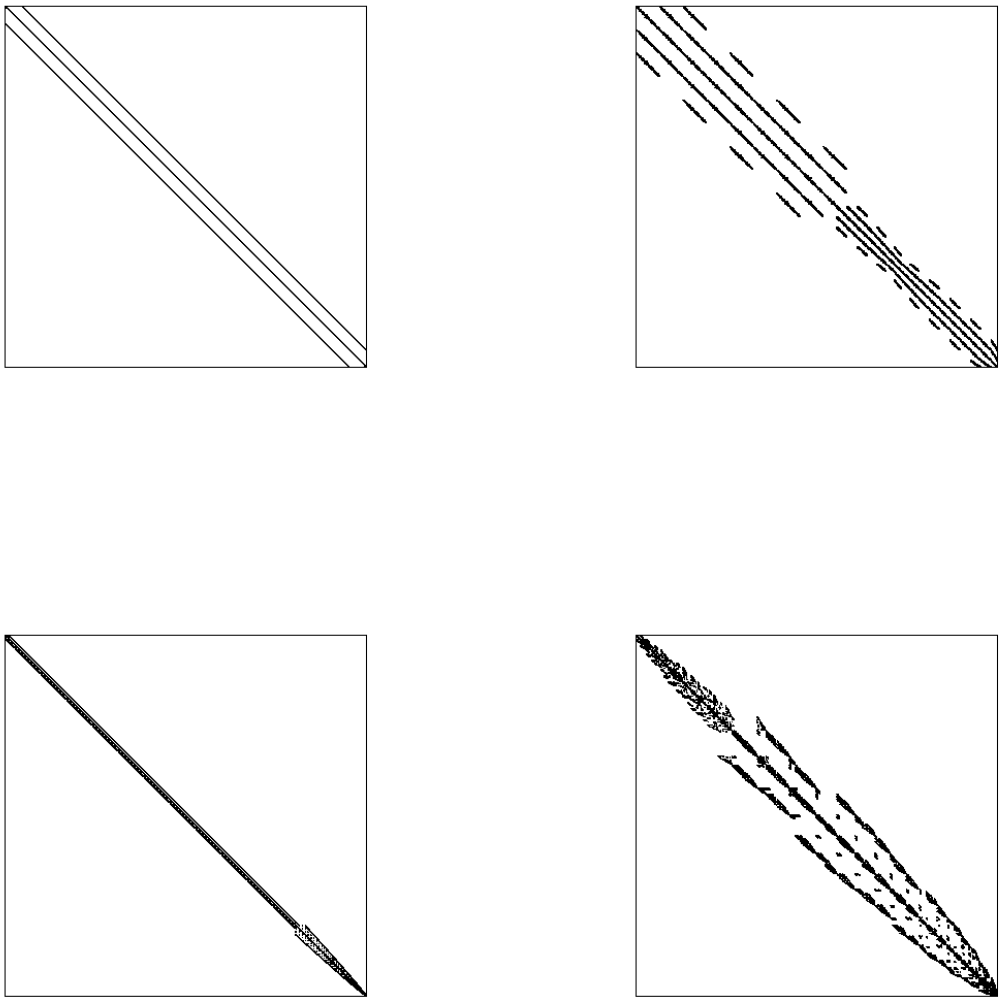

Figure 5.1: Top: sparsity patterns of the matrices from environmental science (left) and structural mechanics (right). Bottom: the same matrices with bandwidth reduction.

Fig. 5.1 displays the sparsity patterns of the FE matrices. The matrix from environmental science has essentially a band structure with a maximum bandwidth of 2375. The matrix from structural mechanics has a much more irregular structure; the maximum bandwidth is 3474.

When the bandwidth of the matrices is diminished, the communication overhead in each iteration of both algorithms is possibly reduced. Since communication is necessary for the operation row-times-vector of the matrix-vector product, a smaller bandwidth results in smaller message length or even in communication with fewer processors. Here, the matrix is reordered by the reverse Cuthill-McKee (RCM) scheme [29], which is a similarity transformation. In FE models, this scheme is frequently used for the assembly of the coefficient matrix; it is performed only once if the mesh does not change, whereas in many cases equation systems or eigenproblems are frequently solved, e.g. in each time step of a time dependent problem.

Fig. 5.1 also shows the sparsity patterns of both matrices with bandwidth reduction. For the matrix from environmental science, the bandwidth is reduced by 45%; the maximum bandwidth is 1303. When the reverse Cuthill-McKee scheme is applied, the maximum bandwidth of the matrix from structural mechanics decreases to 2989; this is a reduction by merely 14%.

In all following tests with respect to eigenvalue approximation, the tridiagonal solver determines the eigenvalues of the tridiagonal matrices with a precision of 11 decimals; the stopping criterion is described in [12]. Simple non-spurious eigenvalues that equal spurious eigenvalues with a precision of 9 decimals are marked as non-isolated simple eigenvalues. The tolerance ϵ_t from (2.2) is set to 10^{-6}.

5.2 Performance Results

On the iPSC/860, the tests were performed using the FORTRAN compiler of the Portland-Group, Inc., release 4.0 and the NX/2 operating system [1] [2]. On the PARAGON, we applied the version 4.5 of the PARAGON FORTRAN compiler and the PARAGON OSF/1 operating system, release 1.1 [3] [4]. The programs were compiled with the optimization switches *-O4 -Knoieee* on both systems.

In the first five investigations, we do not apply bandwidth reduction to the matrices.

In Fig. 5.2, execution times per iteration of the two variants of the CG method and the Lanczos tridiagonalization are compared. With increasing number of processors, the execution times of Algorithms 2.2 and 2.4 are considerably less than those of Algorithms 2.1 and 2.3 because the former variants require only one global communication. On 140 processors, the times are reduced by 28% for the matrix from environmental science and by 23% for the matrix from structural mechanics in the case of the CG algorithm. For the Lanczos tridiagonalization, the corresponding times decrease by 25% and 15%, respectively. Therefore, we only consider Algorithms 2.2 and 2.4 in all following investigations.

On the left, Fig. 5.3 displays execution times per iteration for three different data distributions: $\xi \to \infty$, $\xi \to 0$, and $\xi = 8.3$ for the CG method or $\xi = 2$ for the Lanczos tridiagonalization. For the matrix from environmental science, the execution times are nearly the same since the matrix has a regular structure. In the case of the matrix from structural mechanics, the execution times using the criteria "same number of non-zeros" and "same number of operations" are reduced by ca. 18% compared with the time using the criterion "same number of rows" for both algorithms. Because of the very different number of non-zeros per row, the operations for the computation of the matrix-vector product are not uniformly distributed to each processor applying the latter criterion. The times for the criteria "same number of non-zeros" and "same number of operations" are nearly the same since the operational contribution for the computation of the matrix-vector product

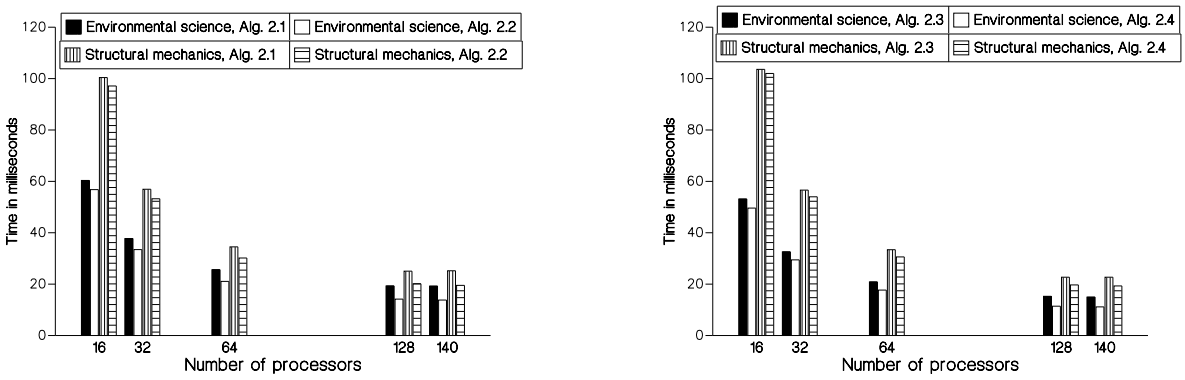

Figure 5.2: Execution times per iteration, PARAGON. Left: two variants of the CG algorithm; right: two variants of the Lanczos tridiagonalization.

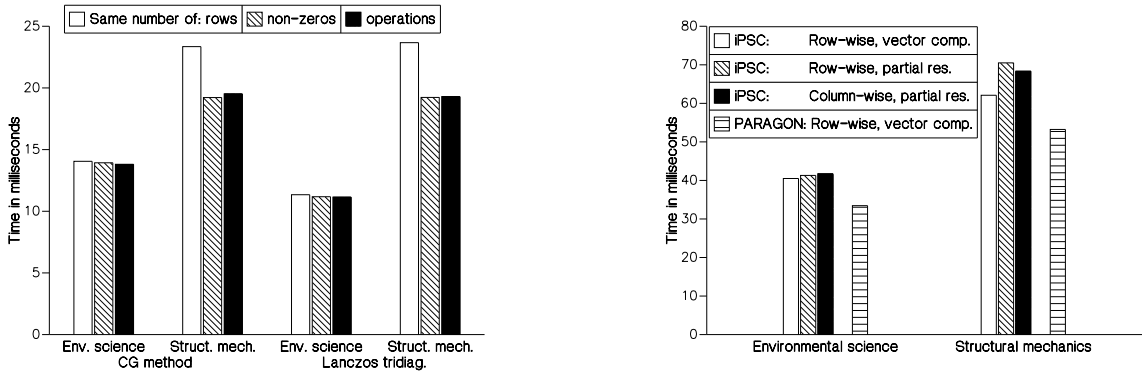

Figure 5.3: Execution times per iteration. Left: different data distributions, PARAGON, 140 processors; right: different communication schemes, CG method, iPSC/860, PARAGON, 32 processors.

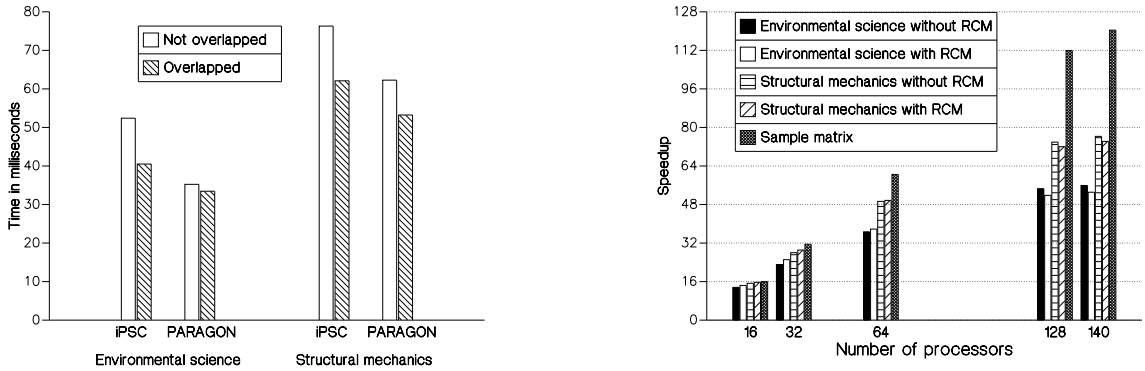

Figure 5.4: CG method. Left: the influence of overlapping, execution times per iteration, 32 processors; right: speedups, PARAGON.

to one iteration is 95% in the case of the CG method and 99% in the case of the Lanczos tridiagonalization. In all following investigations, we apply the criterion "same number of operations".

Fig. 5.3 also shows execution times per iteration for the three considered communication schemes on 32 processors in the case of the CG method. For the matrix from environmental science, schemes 1 to 3 result in almost the same execution times. This is caused by a regular discretization mesh. In the case of the matrix from structural mechanics, the time for the second scheme increases markedly compared with the time for the first scheme. On the one hand, this is caused by a decreased overlap of computation and communication in the case of the second scheme; on the other hand, the new distribution of the data blocks destroys the load balancing because of the irregular structure of that matrix. For the third scheme, the increase of the execution time is smaller since the data blocks are not redistributed. Since schemes 2 and 3 do not result in an improvement compared with scheme 1 for the matrices considered, we apply the latter scheme in all following investigations. The times for the first scheme on 32 PARAGON processors are added to demonstrate the effect of the smaller cycle time of the PARAGON processors i860 XP (20 nanoseconds) compared with the iPSC processors i860 XR (25 nanoseconds). The times on the PARAGON are reduced by ca. 15% compared with the corresponding times on the iPSC.

In Fig. 5.4, times per iteration on 32 processors with and without the overlapped execution of computation and communication are presented for the parallel CG algorithm. On the iPSC, the overlapped execution reduces the execution times by ca. 20% in both cases whereas the overlap results on PARAGON merely in an decrease of the execution times by ca. 5% in the first case and by ca. 15% in the second case. The main reason for the difference is the much higher transfer rate on the PARAGON compared with that on the iPSC.

On the right, Fig. 5.4 shows speedups on 16 to 140 processors for the CG method with and without bandwidth reduction of the matrices. The equation system from environmental science together with the program code and the remaining data requires the memory of more than two processors, that from structural mechanics the memory of more than four processors, and the sample system the memory of more than eight processors. For up to four in the first case, up to eight in the second case, and up to 16 processors in the third case, linear speedup was assumed. Bandwidth reduction results in slightly higher speedups for up to 64 processors. On 128 and 140 processors, the speedups for the matrices with bandwidth reduction are even slightly less than for the matrices without bandwidth reduction. On 140 processors and without bandwidth reduction, the speedup is 55.9 in the first case and 76.3 in the second case. This corresponds to efficiencies of 40% and 55%. With bandwidth reduction, the speedups are 53.1 and 74.2, respectively; the efficiencies are 38% and 53%. For the sample matrix, a speedup of 120.4 is achieved on 140 processors; the corresponding efficiency is 86%. The solution of the sample system with a precision of five decimals on 140 processors requires an execution time of 0.86 seconds.

Fig. 5.5 displays the corresponding speedups of the total Lanczos method for the matrix from structural mechanics with bandwidth reduction and for the sample matrix. The iteration was stopped when at least 100 eigenvalues of the matrix had been determined with the predefined precision. After every 1000 steps of the Lanczos tridiagonalization, the eigenvalues of the generated tridiagonal matrices were computed. In total 2000 steps were necessary to find 136 eigenvalues in the first case and 105 eigenvalues in the second.

On 140 processors, speedups of 67.9 and 123.3 are achieved; this corresponds to efficiences of 49% and 88%, respectively. The total execution times on 140 processors are 45.2 seconds for the matrix from structural mechanics and 82.4 seconds for the sample matrix. The contribution of the time for the tridiagonal solver to the total time is 11% in the first

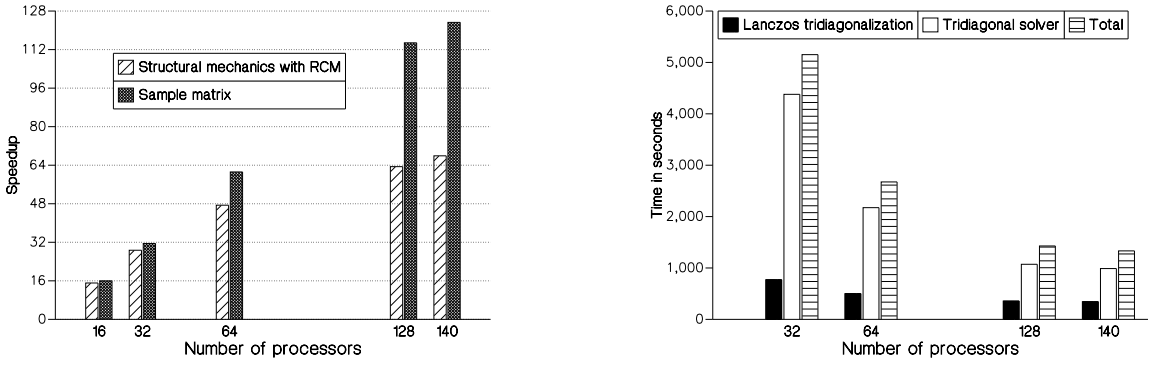

Figure 5.5: Total Lanczos method, PARAGON. Left: speedups; right: execution times, matrix from environmental science with bandwidth reduction.

case and 7% in the second.

Fig. 5.5 also shows execution times of the total Lanczos method for the matrix from environmental science with bandwidth reduction. The same stopping criterion as before was applied. In this case, after every 10000 steps of the Lanczos tridiagonalization the eigenvalues of the generated tridiagonal matrices were determined. In total 30000 steps were necessary to find 449 eigenvalues. Here, the time for the tridiagonal solver is dominant. On 140 processors, the contribution to the total time is 74%. Assuming linear speedup for up to 32 processors, a total speedup of 123.7 was achieved on 140 processors; the efficiency is 88%. Fig. 5.5 demonstrates that both the developed parallel Lanczos tridiagonalization and the tridiagonal solver scale well on distributed memory machines.

6 Conclusions

We presented a parallelization strategy for the iterative solution of both sparse systems of equations and eigenproblems on distributed memory systems and demonstrated by case studies that the developed data distribution and communication schemes, which enable the overlapped execution of local computations and communication, do result in efficient iterative algorithms. These algorithms perform well for large sparse matrices with very different sparsity patterns coming from real finite element applications.

In recent investigations, the parallelization strategy above has been applied successfully to CG methods with polynomial preconditioning [8] and the QMR algorithm for solving non-Hermitian systems of linear equations [18].

References

[1] Intel Supercomputing Systems Division, Beaverton, Oregon. *iPSC/860 Fortran Compiler User's Guide*, January 1993. Order Number: 312131-003.

[2] Intel Supercomputing Systems Division, Beaverton, Oregon. *iPSC/860 System User's Guide*, March 1992. Order Number: 312312-001.

[3] Intel Supercomputing Systems Division, Beaverton, Oregon. *Paragon OSF/1 Fortran Compiler User's Guide*, April 1993. Order Number: 312491-001.

[4] Intel Supercomputing Systems Division, Beaverton, Oregon. *Paragon User's Guide*, October 1993. Order Number: 312489-002.

[5] 3DFEMWATER: a three-dimensional finite element model of water flow through saturated-unsaturated media. Oak Ridge National Laboratory. *ORNL-6386*, 1987

[6] SMART, Benutzerhandbücher. Institut für Statik und Dynamik der Luft- und Raumfahrtkonstruktionen der Universität Stuttgart. *ISD-Berichte*, 1976-1992.

[7] E. Anderson et al. *LAPACK user's guide.* SIAM, Philadelphia, 1992.

[8] S.F. Ashby. Minimax polynomial preconditioning for hermitian linear systems. *SIAM J. Matrix Anal. Appl.*, 12:766–789, 1991.

[9] C. Aykanat, F. Özgüner, D.S. Scott. Vectorization and parallelization of the conjugate gradient algorithm on hypercube-connected vector processors. *Microprocessing and Microprogramming*, 29:67–82, 1990.

[10] V.A. Barker, C. Yingqun. LANSYM: A Fortran Subroutine for Computing Eigensolutions of Symmetric Sparse Matrices on the Connection Machine. Institute for Numerical Analysis, Technical University of Denmark, Lyngby, *Report NI-92-12*, December 1992.

[11] A. Basermann. Conjugate gradients parallelized on the hypercube. *International Journal of Modern Physics C*, Vol. 4, No. 6:1295–1306, 1993.

[12] A. Basermann, P. Weidner. A parallel algorithm for determining all eigenvalues of large real symmetric tridiagonal matrices. *Parallel Computing*, 18:1129–1141, 1992.

[13] L. Collatz. *Eigenwertaufgaben mit technischen Anwendungen.* Akademische Verlagsgesellschaft Geest & Portig K.-G., Leipzig, 1963.

[14] J.K. Cullum, R.A. Willoughby. *Lanczos Algorithms for Large Symmetric Eigenvalue Computations.* Volume I: Theory, Birkhäuser, Boston Basel Stuttgart, 1985.

[15] J.K. Cullum, R.A. Willoughby. *Lanczos Algorithms for Large Symmetric Eigenvalue Computations.* Volume II: Programs, Birkhäuser, Boston Basel Stuttgart, 1985.

[16] N. Feistl. *Das Verfahren der konjugierten Gradienten auf Vektorrechern.* Diplomarbeit, Ludwig-Maximilians-Universität, München, Juli 1990.

[17] P. Fernandes, P. Girdinio. A new storage scheme for an efficient implementation of the sparse matrix-vector product. *Parallel Computing*, 12:327–333, 1989.

[18] R.W. Freund, N.M. Nachtigal. QMR: a quasi-minimal residual method for non-Hermitian linear systems. *Numerische Mathematik*, 60:315–339, 1991.

[19] M.R. Hestenes, E. Stiefel. Methods of conjugate gradients for solving linear systems. *Journal of Research of the National Bureau of Standards*, 49:409–436, 1952.

[20] T.J.R. Hughes. *The Finite Element Method.* Prentice-Hall, Englewood Cliffs, 1987.

[21] S.K. Kim, A.T. Chronopoulos. A class of Lanczos-like algorithms implemented on parallel computers. *Parallel Computing*, 17:763–778, 1991.

[22] C.P. Kruskal, L. Rudolph, M. Snir. Techniques for parallel manipulation of sparse matrices. *Theoretical Computer Science*, 64:135–157, 1989.

[23] O.A. McBryan, E.F. Van de Velde. Matrix and vector operations on hypercube parallel processors. *Parallel Computing*, 5:117–125, 1987.

[24] J.M. Ortega. *Introduction to Parallel and Vector Solution of Linear Systems.* Plenum Press, New York London, 1988.

[25] B.N. Parlett. *The Symmetric Eigenvalue Problem.* Prentice-Hall, Englewood Cliffs, 1980.

[26] S. Pissanetsky. *Sparse Matrix Technology.* Academic Press, London Orlando, 1984.

[27] Y. Saad. *Numerical Methods for Large Eigenvalue Problems.* Manchester University Press, Manchester, 1992.

[28] U. Schendel. *Sparse-Matrizen.* R. Oldenbourg Verlag, München Wien, 1. Auflage, 1977.

[29] H.R. Schwarz. *FORTRAN-Programme zur Methode der finiten Elemente.* B. G. Teubner, Stuttgart, 1981.

[30] H. Vereecken, G. Lindenmayr, A.Kuhr, D.H. Welte, A. Basermann. Numerical modeling of field scale transport in heterogeneous variably saturated porous media. *KFA/ICG-4 Internal Report No. 500393*, January 1993.

[31] J.H. Wilkinson. *The Algebraic Eigenvalue Problem.* Clarendon Press, Oxford, 1965.

[32] J.H. Wilkinson, C. Reinsch. *Handbook for Automatic Computation.* Volume II: Linear Algebra, Springer-Verlag, Berlin Heidelberg New York, 1971.